AIAA
G-010-1993

Guide

Reusable Software: Assessment Criteria for Aerospace Applications

AIAA

AIAA
G-010-1993

Guide for Reusable Software: Assessment Criteria for Aerospace Applications

Sponsor

American Institute of Aeronautics and Astronautics

Abstract

This AIAA Guide provides a basis for assessing potentially reusable software. It introduces the concept of domain analysis and describes the principal products of this method. Criteria for assessing the reusability of software down to the component level, along with specific examples are included. A methodology for storing the analyses and criteria and establishing a reuse library are given.

Guide for reusable software : assessment criteria for aerospace
applications ; sponsor, American Institute of
Aeronautics and Astronautics.
p. cm.
Includes bibliographical references.
ISBN 1-56347-049-7
1. Computer software--Reusability. 2. Aeronautics--Computer
programs. 3. Astronautics--Computer programs. I. American
Institute of Aeronautics and Astronautics.
QA76.76.R47G85 1993
005.1--dc20
93-28289
CIP

Published by
American Institute of Aeronautics and Astronautics
370 L'Enfant Promenade, SW, Washington, DC 20024

CONTENTS

Foreword

This Guide for Reusable Software: Assessment Criteria for Aerospace Applications has been sponsored by the American Institute of Aeronautics and Astronautics as part of its Standards Program.

Software reuse and the development of criteria to aid in assessing potentially reusable software is an emerging discipline. This AIAA Guide provides the reader with information on performing domain analysis as the basis for developing criteria for assessing potentially reusable software and establishing a software reuse library. It includes guidance on performing the following:

- Domain analysis
- Component assessment
- Reuse library assessment.

This Guide was developed to meet the varying needs of software personnel such as:

- Managers
- Software engineers
- Quality engineers
- Software reuse librarians
- Reuse analysts
- Reuse researchers.

The AIAA Standards Procedures provide that all approved Standards, Recommended Practices, and Guides are advisory only. Their use by anyone engaged in industry or trade is entirely voluntary. There is no prior agreement to adhere to any AIAA standards publication and no commitment to conform to or be guided by any standards report.

In formulating, revising, and approving standards publications, the Committees on Standards will not consider patents which may apply to the subject matter. Prospective users of the publications are responsible for protecting themselves against liability for infringement of patents or copyrights, or both.

This project was an undertaking of the AIAA Committee on Standards for Software Systems (Soft/CoS) and its Software Reuse Working Group (SoftReWG). The SoftReWG developed this document and incorporated comments of reviewers from academia, government, and industry. After broader use of the Guide, it is planned to submit it for approval as an American National Standard.

AIAA Software Systems Committee on Standards

At the time of preparation, the following individuals were members of the AIAA Software Systems Committee on Standards:

Beverly Kitaoka, Chairman (Science Applications International Corporation)
John W. Brackett (Boston University)
Ed Comer (Software Productivity Solutions)
Anne B. Elson (Jet Propulsion Laboratory)
Steven Glaseman (Aerospace Corporation)
Frank Lamonica (Rome Laboratory)
Brian Larman (Jet Propulsion Laboratory)
Constance Palmer (McDonnell Douglas Missile Systems Company)
Richard J. Pariseau (Naval Air Warfare Center, Aircraft Division)
Randall Scott (Software Productivity Consortium)
Alice A. Wong (Federal Aviation Administration)

The following individuals constituted the Software Reuse Working Group:

Beverly Kitaoka
Frank Lamonica
Brian Larman

The document was approved by the Software Systems Committee on Standards in March 1993.

The AIAA Standards Technical Council (Ali H. Ghovanlou, Chairman) approved the document in May 1993.

1.0 INTRODUCTION

1.1 Purpose

This Guide provides the reader with information on performing domain analysis as the basis for developing criteria for assessing potentially reusable software and establishing a software reuse library. It focuses on the applicability of this approach within aerospace applications domains. This approach is relevant both to organizations with established reuse programs and to those desiring to establish a software reuse program.

1.2 Scope

This Guide commences with a discussion of domain analysis, followed by component assessment and concludes with the role of a reuse library. Section 2.0 defines domain analysis, and describes the principal products of such an analysis. Discussion of various approaches and techniques documented in the literature is then provided. Examples of current aerospace applications domain analysis are the Common Ada Missile Parts (CAMP) and the Naval Training Systems Center Reuse Initiative projects. Criteria to be considered for domain analysis are then presented.

Section 3.0 explains component assessment for reuse and provides criteria that could be used for such an assessment. To achieve significant productivity gains with reuse, the traditional concept of components as source code must be expanded to include components of many types such as specifications, requirements, designs, data sets, and test sets. The component assessment criteria of section 3.0 are divided into three categories: domain assessment, reuse assessment, and software assessment criteria.

Finally, as presented in section 4.0, the assessment criteria specified during the domain analysis process must be captured and stored in a reuse library where they are readily accessible to both the component developers and the component assessors. The reuse library stores information about the domain and provides tools to facilitate its use. The role of the library with respect to component assessment is described to provide a complete picture of the reuse process from the assessment perspective.

1.3 Intended Audience

This guide is intended for use by both practitioners (e.g., software developers, managers, quality engineers, and reuse librarians) and researchers. Its purpose is to provide a common baseline for discussion and to define a procedure for assessing the reusability of software, performing domain analysis, and setting up a reuse library.

1.4 Background

As the cost of software development rises, techniques for controlling the upward trend must be pursued and implemented. One of the most promising techniques being employed involves reusing software that has been developed and paid for in prior projects. The use of the Ada language also assists in controlling the life cycle costs of software development. In addition to being a Department of Defense (DoD) requirement since 1987, there are many compelling reasons for using Ada in today's aerospace applications. Coupled with effective software design methods, such as object oriented design, Ada directly supports the development of highly reliable and maintainable aerospace systems. It provides for an efficient development environment, particularly when coupled with the concept of reusability, thus offering the potential for significant productivity improvements. Ada also supports software portability and reuse while standardizing on a single development environment for component development.

With few exceptions, the reuse of software has not been overly successful to date. In the past, software has typically been developed to fit a particular function with no thought to enhancing the component for use on another project. Software has not been designed with reuse as a primary objective. Software must be designed and developed with reuse as a primary goal in order for components to be reused effectively in other projects or applications.

1.5 The Component Assessment Process

The main objectives of the assessment process are to provide guidance to software component developers in producing good reusable components and to assist the assessor in evaluating these potential reusable components. To aid in the automation of the assessment process, several commercial off-the-shelf (COTS) tools are currently available. To aid in the assessment of domain specific applications, corresponding domain specific assessment tools are also being developed.

Reuse will have a major impact on the classical software life cycle if reuse is an integral part of each life cycle phase and if special consideration and concerns are addressed. To achieve the productivity gains envisioned, tasks such as searching for existing products which meet the system requirements either directly or through simple modification should be included in the project proposal stage. In each of the subsequent phases of the software life cycle there are corresponding places for inclusion of reuse (e.g., rapid prototyping, requirements analysis, and testing) in addition to actual software development.

If the suppliers of reusable software components have performed the domain analysis (see section 2.0) correctly, without a bias towards a specific architecture or proprietary product, then these components should be appropriate for most users. This underscores the critical need to have a skilled component assessor perform assessments so that many appropriate components are placed in the library.

1.6 Assumptions

In developing the assessment criteria, the following assumptions were made. Domain analysis personnel and component developers, if not the same, must work together to develop reusable components. In addition, component developers and the assessor must have access to the assessment criteria during the development of the reusable components. Without some guidance, the developed components will not meet these criteria. In addition, assessment tools based on the assessment criteria must be available to assist the component developers and component assessors in the assessment task so that it does not become overwhelming.

The concept of developing reusable components is new to many component developers, as are the assessment criteria. There are many lessons to be learned which will result in refining these criteria. As a reuse library becomes populated with components, users will identify good reusable components by the number of accesses, extractions, and user comments. This data and other metrics should be used to refine the assessment criteria.

2.0 DOMAIN ANALYSIS

Domain analysis is an approach to the analysis and structuring of software requirements aimed at facilitating reuse of software requirements, design, code, and test information across a domain. Performing a domain analysis entails factoring out commonality and factoring in generality. A domain is a set of problems with similar requirements for which a general solution can be developed.

Examples of aerospace applications domains are identified below:

Control systems - Systems typically characterized by stringent timing requirements, processor and memory resource limitations, and safety / fault tolerance requirements. Applications include systems for the control of aircraft, missiles, satellites, and industrial processes.

Signal processing systems - Systems that analyze and respond to complex signals, typically characterized by complex processing algorithms on high performance hardware. Applications include electronic warfare, intelligence signal processing, and satellite image processing.

Command and control systems - Systems that are typically characterized by significant man-in-the-loop involvement, complex user interfaces and data bases, and coordination of distributed functions. Applications include military C^3I systems, communications network control systems, and ground control systems for satellites.

The scope of a domain is typically set through consideration of economic and technical factors associated with system implementation and the business objectives of the organization. As such, the principal determination of a domain is whether software artifacts developed for one instance of the domain may be cost-effectively reused for another instance.

The principal products of a domain analysis, relevant to the assessment process, are:

Domain Definition - a working definition of the domain that provides sufficient information to determine whether a specific system is a candidate for inclusion in the domain. The domain definition characterizes the functional boundaries of the domain and helps determine whether it will pay to reuse existing domain components in a new system. This boundary may change throughout the domain analysis process (and beyond) as key economic and technical issues associated with domain software reuse become more clearly understood. The key determinant of this boundary is the potential for cost-effective reuse of software components. After a software reuse program has been implemented, the domain definition can be further refined based upon various usage metrics which can be collected from suppliers and users.

Conceptual Taxonomy - a definition of domain terminology that provides an initial basis for relating domain instances. The taxonomy includes any necessary thesaurus information to begin the process of correlating different instances of the domain. It also includes structuring information to aid in the understanding of how domain terms relate. This is also often referred to as a domain vocabulary.

Canonical Requirements Model - a decomposition of the domain according to criteria that are stable with respect to potential variations among instances, along with descriptions of services, performance criteria, constraints, and potential variations. Canonical requirements are the most important product of domain analysis. They provide a framework from which implementations of specific instances may proceed without recreating existing requirement information.

Tracz (Tracz 1987) states that: To answer the question, "What software should be made reusable?" it is helpful to rephrase the question into two separate questions:

1. "What software is common among most applications?" and

2. "What software is common within a specific application domain?"

This provides two categories of software which seem to be good candidates for reuse: horizontally reusable and vertically reusable components. Horizontal reuse refers to reuse across a broad range of application areas (such as data structures, sorting algorithms, and user interface mechanisms). Vertical reuse refers to components of software within a given application area that can be reused in similar applications within the same problem domain. Horizontal reuse has been studied the most to date (Booch 1987), and it likely has occurred much more frequently than vertical reuse. The main reasons for this are that horizontal reuse is better understood and easier to achieve. On the other hand, a great potential leverage can come from vertical reuse - by intensive reuse of carefully crafted solutions to problems within an application domain. The Naval Training Systems Center (NTSC) Reuse Initiative project discussed in 2.1 is an example of vertical reuse.

2.1 Domain Analysis Approaches

As with any relatively new concept, the methods used vary widely. Each new project or team develops a new approach based upon existing / previous methods modified with their better idea. Domain analysis is no exception. Currently, there is not an established technique for performing domain analysis. However, three aspects that most of the methods have in common are:

- producing a domain-specific vocabulary;

- partitioning the domain into more manageable sub-domains; and
- creating a domain model.

Areas where the methods vary include:

- the type of persons responsible for conducting domain analysis;
- variation in the means for representing their findings; and
- the use of the products.

There are a number of domain analysis methods that have been documented in the literature. This section briefly describes the approach currently being implemented in one aerospace applications domain - the NTSC Reuse Initiative. Excerpts from other approaches that may be of interest to the reader will follow.

The NTSC Reuse Initiative project began with a domain analysis of the flight simulation domain that examined four simulators to identify common classes of objects used in flight simulation. The simulators examined were the V-22 Operational Flight Trainer, UH-1 Flight Simulator, P-3A/B Tactical Navigation Modernization Operational Flight Trainer, and C-17A Weapons System Trainer. The domain analysis methodology used under the NTSC Reuse Initiative was based on the process developed by Dr. Ruben Prieto-Diaz and consists of the following steps:

1) Define the flight simulation domain through top-down object-oriented analysis.

- Collect domain information and data.
- Identify domain bounds (for this project, the customer specified flight simulation, Ada, OOD).
- Identify the sample base (i.e., C-17, V-22, P-3A/B, UH-1).
- Perform an object-oriented analysis to identify the objects within the selected sub-domains.

2) Classify the flight simulation domain entities through bottom-up analysis.

a) Identify the objects and operations from the sample base.

- Analyze the existing design.
- Analyze the requirements.
- Extract the descriptors.
- Decompose the statements by keywords.

b) Abstract and classify the flight simulation data.

- Group terms.
- Name clusters.
- Arrange by facets.
- Define standard classification templates.
- Construct the thesauri.

3) Derive the flight simulation models by consolidating the top-down and bottom-up approaches.

- Group the descriptors into classes.
- Use the class structure to create a generic model.
- Expand the models and classification.

4) Expand and verify the selected models and classifications.

The objective of this domain analysis was to produce a set of domain models which will support the creation and reuse of software objects for the NTSC Reuse Initiative. This in turn supports reuse-based software development. Domain models may range in complexity from simple definitions of the domain in terms of common requirements to a detailed taxonomy and classification scheme or an elaborate generic architecture. The domain model produced for the NTSC Reuse Initiative was a detailed taxonomy and classification scheme.

Other approaches to various aspects of domain analysis and related terminology are described below.

Kang (1989) briefly summarizes an approach to domain analysis:

Domain analysis is an activity to produce a domain model, a dictionary of terminologies used in a domain, and a software architecture for a family of systems. The objectives of a domain analysis are:

- facilitate reuse of domain knowledge in systems development;
- define the context in which reusable components can be developed and the reusability of candidate components can be ascertained;
- provide a model for classifying, storing, and retrieving software components;
- provide a framework for tooling and systems synthesis from the reusable components;
- allow large-grain reuse across products; and
- identify software assets.

Based on Kang's experience with domain analysis (called features analysis in this project) and the potential benefits from it, he believes that domain analysis should be a standard activity in the software development life cycle.

Lee and Rissman (1989) describe their work at the Software Engineering Institute (SEI) in determining domain-specific software architectures.

Prieto-Diaz (1987) comments on the need to define a domain boundary (i.e., where one domain ends and another begins). He views the domain-specific language as encapsulated in a formal language and serving as a specification language for the construction of systems in the domain. Prieto-Diaz characterizes this as the "reuse of analysis of information," and states the opinion that this "is the most powerful sort of reuse."

Prieto-Diaz also briefly summarizes the domain analysis approaches used by Raytheon (in the work reported by Lanergan and Grasso [1984]), and by McDonnell Douglas (in the Common Ada Missile Packages [CAMP] work).

Neighbors (1987) reports on the deliberations of the Domain Analysis Working Group at the Workshop on Software Reuse, held in October 1987. The report states that "given a domain analysis, an organization should be able to: (1) use the domain model to check the specifications and requirements for a new required system in the domain; (2) educate new people in the organization providing them with the general structure and operation of systems in the domain; and (3) derive operational systems directly from the statement of the system in domain specific terms."

The AIAA Software Reuse Working Group undertook the domain analysis of library management systems as a practical problem, with the same individuals serving as both domain experts and domain analysts. Neighbors (1987) describes the group's activities in some detail and concludes by giving the following Basic Domain Analysis Process.

1) Establish the domain subject area.
2) Collect the domain experts.
3) Establish the depth of analysis (i.e., whether to analyze sub-domains).
4) Establish the width of analysis (i.e., determine the boundary of the domain - "Is this function required by most of the systems built in this domain?").
5) Define the specific domain objects, operations, relationships, and constraints.
6) Hand test the domain by attempting a description of a specific system in the domain.

7) Package the domain for constructive reusability by expressing it in a form for a transformational refinement tool.

The working group members used various analysis representations, including data flow diagrams, entity-relationship diagrams, semantic nets, object diagrams, and class hierarchies with inheritance. They concluded that usually only one each from the object hierarchy, data flow, and control flow representations would be needed.

Hutchinson and Hindley (1988) report on their work in developing a domain analysis method. Their goals were:

- to discover the functions that underwrite reusability;
- to focus the domain specialist's attention on reuse;
- to help the domain specialist ascertain reuse parameters;
- to discover how to redesign existing components for reuse; and
- to organize any domain for reuse.

The domain analysis was performed by a reuse analyst with the assistance of a domain specialist, an individual with a full understanding of the problem domain. The researchers developed structured domain analysis techniques based on questions devised to assess a software component's reusability. The domain on which they based their experimentation was a simulation of the utility systems management (USM) system of the Experimental Aircraft Programme (EAP) in the United Kingdom. The sub-domains they considered were propulsion, fuel management, and undercarriage. In the case of propulsion, this sub-domain was considered for reuse because the controlled hardware (the engines) would not change significantly between the EAP implementation and the next project. Fuel management was chosen because the domain appeared to contain considerable functional duplication within the requirements definition. Undercarriage was chosen because much of its operation would not change on future implementations.

The reuse analyst decided on three levels of reuse to clarify the domain: the initial level pertained to reuse of the whole system, the next to reuse of subsystems, and the final to functions at the requirements phase and to components at the design and code phases. The reuse analyst presented questions to the domain specialist, based on the assumption that it is domain-specific knowledge that can isolate reusable components. The questions seek to elicit identification of reuse attributes and reusable components in an understandable manner. These questions are contained in a list of domain analysis criteria in section 2.2.

The authors observe that reuse proved to be practical, even in the hardware-dependent areas being analyzed. They assessed the requirements functions as potentially 75 percent reusable for the next implementation, and indicated that reuse could be equally high for code designed for reuse from these requirements.

2.2 Domain Analysis Criteria

Using the documented domain analysis activities summarized in section 2.1, a preliminary set of criteria for domain analysis was developed. These criteria can be used by the reuse analyst to determine the potential reusability of components, to identify classes of objects, and to describe the domain-specific vocabulary. The criteria in appendix A are oriented specifically to eliciting data from domain experts for reuse applications. These criteria should help determine what software should be made reusable.

3.0 COMPONENT ASSESSMENT

Reuse may occur at any and all stages in the software development life cycle, resulting in benefits ranging from minuscule to considerable. Reusable components may alter only the implementation of a software development effort or they may impact the design.

The most valuable tools and components will alter the entire scope and structure of software development efforts in which they are used, saving large quantities of time for engineering, development, and testing.

Good reusable software provides needed functionality and is substantially less expensive to reuse than to recreate. For small components like primitive math functions and data structures, the cost of reuse depends heavily on the quality of the reusable component. Components which use meaningful naming conventions and reasonable coding standards are more easily understood and less expensive to reuse. Since the software may have to be modified, the component developers should be able to examine the code and make changes without introducing errors or triggering undocumented side effects.

As components increase in size, the cost of reuse becomes more closely tied to the cleanness of the abstractions supported by the reusable components and to the amount of interface and code documentation provided with the components. Ideal components would, as a minimum, include: design documents, code, test procedures, test data, and test results. Modules whose data types and entry points are tightly cohesive are more easily understood. Modules which are loosely coupled are more easily reused. Reusable components should avoid hidden side effects, should not reference global data structures or hardware devices that may not be present when it is reused, and should export clear, concise, cleanly abstracted interfaces.

The assessment criteria in Appendix B support the verification of a single or a small set of related properties for reuse.

These criteria fall into the following categories:

- Domain Assessment
- Reuse Assessment
  - Internal Documentation
  - Machine Dependencies
  - Logic and Functionality
  - Avionics Domain
- Software Assessment
  - Identifiers
  - Local Programming Practices
  - Tasking Semantics
  - Application of Generics
  - Global Programming Practices

4.0 REUSE LIBRARY

Once constructed, components must be classified and placed in a reuse library designed to minimize the time and effort required to find and extract potentially reusable components. The apparatus for storing and managing software contributes greatly to its usability. That apparatus should include a software library and an intelligent scheme for classifying software so that user searches are successful in finding components that match, or if there is not an exact match, then identifying components whose characteristics are close to those attributes specified. To facilitate browsing through an unfamiliar library, software should be designed, classified, and retrieved using a vocabulary that is appropriate to the set of objects and operations manipulated, the set of services provided, and the domain of anticipated reuse.

Criteria that should be considered when evaluating alternative library systems are contained in Appendix C.

4.1 Emerging Issues - Software Reuse Library Support for a Reuse Process

A software reuse library capable of supporting a reuse process is essential to realize the full potential of reuse (Kitaoka 1990). In this library we must place an expansive set of software life cycle work products which includes software architectures, specifications, designs, data, code, tests, and other useful information.

Reuse of this information will only occur if reuse is incorporated into all activities of the software life cycle process with ready access to the work products which can be accomplished with a software reuse library.

Reuse can be incorporated into most software life cycle processes and a software reuse library should be able to support the reuse process. A software reuse library can support the reuse process in areas such as domain analysis, assessment of potential library components, prototyping, design, implementation, configuration management, quality assurance, and maintenance.

After performing domain analysis and generating a profile of software reuse library users, the software reuse library interface can be tailored to reduce the effort of users by providing repository capabilities which support the activities defined for their respective roles in domain technology. To provide a user-friendly and efficient interactive interface, for example, the library retrieval mechanism should be based on the classification taxonomy produced during domain analysis, and present the access capabilities in the relevant terminology supplied by the domain analysis process.

To assist in reusing prototypes, the software reuse library can store guidelines for reusing prototype components. To allow the exchange of graphic design representations among design tools, the software reuse library can store information about the graphics design representation so the user can select the proper tools for analyzing the design. As components are submitted to the library, assessment tools integrated into the library can assist quality assurance in determining the reusability of the component. The library can also support configuration management with an integrated configuration control system to track component revisions and insure the appropriate construction of compound components.

The software reuse library is essential to the support of a software reuse process and must become an integral part of the software support environment. The issues involved in supporting reuse in software systems development are applicable to many of the software life cycle processes in use today. The success of a particular repository implementation is the extent to which the repository activities in each activity of the reuse process will reinforce software reuse activities as the software engineer uses the environment to create and support the software in a system.

4.2 Emerging Issues - Library Interoperability

In the current state of the art, a reuse library is hosted on a single platform and is not in any way integrated with reuse libraries on other platforms. This lack of integration creates an impediment to reuse since reuse library users cannot conveniently access information stored in multiple libraries. To overcome this impediment, the Software Technology for Adaptable, Reliable Systems (STARS) Reuse Concepts (CONOPS) document proposes seamless library interoperability: the creation of a truly seamless environment for library users, in which the boundaries between libraries are transparent to the user and a convincing illusion of a single library is formed.

The STARS contribution to library interoperability is the Asset Library Open Architecture Framework (ALOAF). The ALOAF provides a standard format for interchanging data among asset libraries and a standard service interface to support the capabilities of asset library systems. These standards form a foundation for library interoperability.

The STARS Demonstration Asset Library System (DALS) will be a prototype library system used to support the STARS Library Interoperability Demonstration. DALS will use ALOAF services and the ALOAF data format to demonstrate nearly seamless library interoperability. Although a DALS user will be aware that libraries exist on different platforms, that user will be able to access any remote library maintained by a DALS implementation conveniently.

From a functional perspective, DALS will provide the user with the ability to browse a catalog of reusable software assets. Depending upon the toolset provided by a particular

DALS implementation, users will be able to see either a graphical or textual view of the catalog. DALS will also provide the user with the ability to import data from and export data to an external file. This capability will support asset interchange among asset libraries managed by DALS implementations.

5.0 CONCLUSION

Reuse of software components is one of the most viable approaches to reducing the escalating costs of software development. Identifying and classifying these components so that they can be easily identified and accessed by software developers is key to the successful implementation of a reuse approach. The methods for classifying components into a library can vary significantly. One method, domain analysis, has been used to specify a vocabulary and model as the framework of a reuse library.

This Guide recommends that domain analysis be performed when implementing reuse. There are various approaches to performing a domain analysis which have been provided in a brief survey; however, no single approach is recommended. It further recommends criteria to be used as a guide during domain analysis.

Once domain analysis has been completed, and the software developer has identified candidate components, the components need to be assessed against both general and domain-specific criteria. A set of criteria to be used as a baseline for assessing domain fitness, reuse, and software is also provided. The results of this assessment, assessment criteria, and the assets themselves can be stored in a reuse library to reduce further the cost of identifying and analyzing reusable components.

6.0 GLOSSARY

Asset: Any unit of information of current or future value to a software-intensive systems development and / or PDSS enterprise. Assets may be characterized in many ways including as software-related work products, software subsystems, software components, contact lists for experts, architectures, domain analyses, designs, documents, case studies, lessons learned, research results, and seminal software engineering concepts and presentations

Asset Library: A collection of software assets controlled by an asset library system. Typically, asset libraries are implemented using an asset library system, which is a computer-based system designed to facilitate the reuse and sharing of software assets. Asset libraries provide a set of services that support qualifying, reusing, and managing software assets. See the Asset Library Open Framework reference for a discussion of these services.

Component: One of the parts that make up a software-intensive system. A component may be hardware or software and may be subdivided into other components.

Domain: An area of activity or knowledge. Domains can be characterized as application, solution, horizontal, or vertical.

- **Application Domain**: The knowledge and concepts that pertain to a particular computer application. Examples include battle management, avionics, C3I, and nuclear physics.
- **Horizontal Domain**: The knowledge and concepts that pertain to the particular functionality of a set of software components that can be utilized across more than one application domain. Examples include user interfaces, database systems, and statistics. Most horizontal domains can be organized as a set of equivalence classes where the distinguishing characteristics are software decomposition style (functional, object-oriented, data-oriented, control-oriented, or declarative), conceptual underpinning, and/or required hardware. One example is subdividing user interfaces into terminal supporting versus bit-mapped, mouse input supporting.
- **Vertical Domain**: The knowledge and concepts that pertain to a particular

application domain. Vertical domains can be organized as a hierarchy of subdomains that specialize the knowledge and concepts as one moves from the root to the leaves. Most application domains can be organized as into a vertical domain hierarchy.

Domain Analysis: The process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest.

Domain Engineering: The construction of components, methods, and tools and their supporting documentation to solve the problems of system / subsystem development by the application of the knowledge in the domain model and software architectures.

Domain Model: A definition of the functions, objects, data, and relationships in a domain.

Domain-specific Language: A machine processable language whose terms are derived from the domain model and that is used for the definition of components.

Framework: A skeletal structure to support or enclose something. The skeletal structure in these reuse documents is a conceptual structure that delimits the concepts being discussed, supports understanding and technical transition, and promotes evolution.

Life Cycle: All the activities an asset is subjected to from its inception until it is no longer useful. A life cycle may be modeled in terms of phases, which are often characterizations of activities by their purpose or function such as design, code, and test.

Process: A series of steps, actions, or activities to bring about a desired result.

Process Definition: An instantiation of a process design for a specific project team or individual. It consists of a partially ordered set of process steps that is enactable. Each process step may be further refined into more detailed process steps. A process definition may consist of (sub-) process definitions that can be enacted concurrently.

Reengineering: The process of examining, altering, and reimplementing an existing computer system to reconstitute it in a new form.

Reverse Engineering: The process of analyzing a computer system to identify its components and their interrelationships. Same as design recovery.

Reuse: The transfer of expertise. In software engineering, reuse often refers to the transfer of expertise encoded in software related work products. The simplest form of reuse from software work products is the use of subroutine / subprogram libraries for string manipulations or mathematic calculations. The simplest form of reuse of expertise not represented in software work products is the employment of a human experienced in the desired endeavor.

Reuse Engineering: The application of a disciplined, systematic, quantifiable approach to the development, operation and maintenance of software with reuse as a primary consideration in the approach.

7.0 REFERENCES

Booch, G. *Software Engineering with Ada.*, Benjamin/Cummings Publishing Comp., Inc., Menlo Park, CA, 1987

Cohen, J. "GTE Software Reuse for Information Management Systems," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, PA, July 1989.

Common Flight Simulator Components, Prepared for the NTSC Reuse Initiative Project, Contract No. N61339-90-C-0135, SAIC, Orlando, FL, August 1991.

Flight Simulation Domain Vocabulary, Prepared for the NTSC Reuse Initiative Project, Contract No. N61339-90-C-0135, SAIC, Orlando, FL, August 1991.

Hooper, J. W. and Chester, R. O. *Software Reuse Guidelines*, Prepared for the U.S. Army Institute for Research in Management Information, Communications and Computer Sciences, Oak Ridge, TN, December 1989.

Hutchinson, J.W. and Hindley, P.G. "A Preliminary Study of Large-Scale Software and Re-use" in *Software Engineering Journal*, Vol. III, No. 5, pp 208-212, September 1988.

Informal Technical Report for the STARS, *Asset Library Open Architecture Framework*, February 1992.

Informal Technical Report for the STARS, *STARS Reuse Concept*, February 1992.

Kang, K. C. "Features Analysis: An Approach to Domain Analysis," in *Proceedings of the Reuse in Practice Workshop,* eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, PA, July 1989.

Kitaoka, B. J. "Repository Support for a Reuse Process," presented at the Eighth Annual National Conference on Ada Technology, Atlanta, GA, March 1990.

Lanergan, R. G. and Grasso, C.A. "Software Engineering with Reusable Design and Code," *IEEE Trans. on Software Engr.*, IEEESE10(5), 498-501, September 1984.

Lee, K. J., and Rissman, M. "Application of Domain-Specific Software, Architectures to Aircraft Flight Simulators and Training Devices," in *Proceedings of the Reuse in Practice Workshop,* eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, PA, July 1989.

Neighbors, J. M. "Report on the Domain Analysis Working Group Session," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, Boulder, CO, October 1987.

Peterson, A. S. "Coming to Terms with Software Reuse Terminology: a Model-Based Approach," in *ACM SIGSOFT, Software Engineering Notes*, Vol 16, No. 2, pp. 45-51 , Software Engineering Institute, Pittsburgh, PA, April 1991.

Prieto-Diaz, R. *STARS Reuse Library Process Model*, SAIC, Orlando, FL, March 1991.

Prieto-Diaz, R. "Domain Analysis for Reusability," in *Proceedings of Compsac 87*, pp. 23-29, Tokyo, Japan, October 1987.

Tracz, W. "RECIPE: A Reusable Software Paradigm," in *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, pp. 546-55 eds. B.D. Shriver and R. H. Sprague, Jr., Kailua-Kona, HI, January 1987.

Tracz, W. "Software Reuse Rules of Thumb," in *The Software Practitioner*, Vol I, No. 2, pp. 8-11, published by Computing Trends through the State College of Pennsylvania, March/April 1991.

APPENDIX A
DOMAIN ANALYSIS CRITERIA

A1.1 Is component functionality required on future implementations? (Determine what is common between current versions and future applications.)

A1.2 How common is the component's function within the domain? Is there duplication of the component's function within the domain? Items to consider:

a) common implementation language;

b) written for the same operating system;

c) uses the same database system;

d) has the same user interface; and

e) works on the same hardware platform.

A1.3 Is the component hardware dependent? If so, does the hardware remain unchanged between implementations? Can the hardware specifics be removed to another component? (Highly desired)

A1.4 Is the design sufficiently optimized for the next implementation? (Desired, but not mandatory)

A1.5 Can we parameterize a nonreusable component so that it becomes reusable? (Mandatory)

A1.6 Is the component easily tailorable? (Highly desired)

A1.7 Can the design / code be feasibly modified to make it reusable? Items to consider:

a) Can operations that work on the same kind of data be grouped?

b) Can global data be eliminated or encapsulated in modules along with the operations that manipulate it?

c) Can implementations be separated from interfaces (program families)?

d) Can algorithms be generalized to work on different:

- hardware,
- operating systems,
- I/O devices,
- user interfaces, or
- data structures / data bases?

e) Can virtual interfaces be defined to separate:

- hardware,
- operating system,
- I/O,
- user interfaces, or
- data structure/database dependencies?

A1.8 Can a nonreusable component be decomposed to yield reusable components? How valid is component decomposition for reuse? (Consider modification issues above)

A1.9 Is a standard software architecture defined that shows how the component relates to one another in the system? Elements of a standard architecture definition include:

a) static decomposition,

b) dynamic behavior, and

c) interfaces.

A1.10 Is there a classification of the application domain that characterizes the component? (Highly desired, especially if reusable component is to be placed in a reuse library)

A1.11 Is the interface of the component well defined, including its relation to other application software? The interface definition for the component includes:

a) purpose of the interface(s);

b) data formats and flows interaction protocols;

c) fault handling; and

d) physical and hardware configuration.

A1.12 Are the requirements and limitations that the component places on the system well defined? System limitations to be considered are:

a) hardware (computing / sensor / control) constraints;

b) adherence to specific protocols;

c) use of a generic architecture;

d) requirements for other components / software;

e) timing / sizing / size / weight / power constraints; and

f) exceptions / fault handling.

A1.13 Are the adaptation requirements clearly defined? Adaptation requirements to be considered are:

a) flexibility in operations;

b) mission adaptation;

c) environments / site adaptation;

d) platform adaptation;

e) user adaptation; and

f) technology adaptation including:

   1) hardware technology

   - computing architecture
   - computing capabilities (size, weight, power, speed, memory)
   - communications
   - sensors
   - control mechanisms
   - interface standards

   2) software technology

   - languages
   - tools
   - architectures
   - interface standards.

A1.14 Does the adaptation mechanism(s) provided by the component meet the stated adaptation requirements? (Highly desired)

A1.15 What system requirements has the component been engineered to meet? (Documentation should state the requirements this component satisfies.)

A1.16 What requirements / qualities have been emphasized? What has been compromised as a result? (Documentation should state if portability or efficiency has been stressed.)

A1.17 What analysis has been performed to guarantee the component meets these requirements? Aerospace software qualities or characteristics to be considered are:

a) real-time requirements;

b) embedded environments;

c) detailed algorithmic requirements;

d) severe efficiency (time and space) requirements;

e) fault tolerance requirements;

f) survivability requirements; and

g) security requirements.

APPENDIX B
COMPONENT ASSESSMENT CRITERIA

B1.1 Domain Assessment Criteria

Domain criteria focus on the extent to which a particular component satisfies the profile of the domain under study.

- Does the component actually belong to the domain?
- To what extent does the component fit into the domain?
- To what extent does the component fit into the domain architecture?
- To what extent does the component fit into the generic requirements?

B1.2 Reuse Assessment Criteria

Reuse criteria focus on the potential value of a component to other applications / projects.

B1.2.1 Internal Documentation

Description

The source text is inspected by the assessors who are checking for compliance with comment and documentation requirements. Spelling and grammatical errors, layout of headers, copyright notices, descriptions of what the part does, and implementation and machine dependencies are the types of things checked.

Items

a) Is there a description of what the reusable part does? Does the description conform to standards? The description should explain what the part does, not how or why it does it. The description should contain, as a minimum, a purpose, inputs, outputs, and processing. This description should be in the specification header.

b) Are the following documented?

- Author (Optional)
- Date (Optional)
- Copyright Notices (Mandatory)
- Name of Product (Mandatory)
- Project Name (Optional)
- Revision History (Mandatory)

c) Is the spelling correct in the header and all comments? (Spelling errors must be corrected before the component is placed in the library.)

d) Is the grammar correct in the header and all comments? (Grammatical errors must be corrected before the component is placed in the library.)

e) Is there a description of how to use the reusable part? Does it conform to standards? (As a minimum, the description should contain enough information for the user to be able to use it without having to look at the source code.)

f) Is there a comment documenting assumptions made by the reusable part? (None is acceptable.)

g) Are features that are likely to change documented? (None is acceptable.)

h) Are all conditions under which an exception is raised documented (by subprogram declaration)? (None is acceptable.)

i) Is the analysis and derivation of numerical aspects of the reusable part documented? (As a minimum, the description should contain enough information for the user to be able to use it without having to look at the source code.)

j) Are regression tests and their expected results provided for the reusable part? If not, is there a section that explains why? (It is strongly suggested that all components have accompanying tests and expected results before being accepted into the library.)

k) Is the testing and other verification of the reusable part documented? (Mandatory)

l) Is whether the reusable part can be used in a concurrent processing environment documented? (Optional)

m) Are any dependencies on other units documented? (None is acceptable.)

n) Is the basis for design decisions documented? (An ideal component would have its original software development folder which would contain this information.)

o) Are all known constraints on how the code can be changed documented? (Desirable, but not mandatory.)

p) Is the extent to which the behavior of the reusable part is affected by known compiler implementation dependencies documented? (It is highly desirable that all known compiler implementation dependencies be removed.)

q) Are all known constraints that must be satisfied by the parameters documented? (Highly desirable if these constraints can be eliminated.)

r) Are the checks that must be / are made for invalid use of the reusable part documented?

- Checks made by the reusable part.
- Checks that must be made by the client of the part.

s) Does the implementation primarily conserve space or conserve time?

B1.2.2 Machine Dependencies

Description

In this section, anything that is dependent on the machine or avoids dependence on the machine is inspected. In general, all of the following items should be enforced.

Items

a) Have rules been included that list the accuracy limits of the target machines? Are programming practices such that they do not press these accuracy limits?

b) Does the reusable part assume no more and no fewer than 16 bits for INTEGER types?

c) Does the reusable part assume no more and no fewer than 6 decimal digits of precision available for floating point types?

d) Does the reusable part assume no more and no fewer than 32 bits available for fixed point types?

e) Does the reusable part assume no more and no fewer than 72 characters per line of source text?

f) Does the reusable part assume no more and no fewer than 16 bits for universal integer expressions?

g) Does the reusable part assume that the range of DURATION is -86_400 .. 86_400 seconds?

h) Does the reusable part assume that the value for DURATION'S MALL is 20 milliseconds?

i) Are representation clauses used to specify the collection size for access types? This is an acceptable exception to the guideline to minimize the use of representation clauses.

B1.2.3 Logic and Functionality

Description

The items in this section are used as a guide for assessors who are checking the logic and functionality of the reusable part. These items require judgment calls on the part of the assessor.

Items

a) Is complete functionality provided in a reusable part or set of parts?

b) Are generic units built to anticipate change? Do generic units use generic parameters to facilitate flexibility?

c) Is Ada's generic construct exploited to create readily adaptable parts that can be instantiated to provide specific functionality using generic subprogram parameters?

d) Does the functionality provided match that described?

B1.2.4 Avionics Domain

Description

The items in this section are used as a guide for assessors who are checking avionics domain-specific aspects of the reusable part. These items require domain expertise on the part of the assessor.

Items

a) There are certain instances where it is desirable to use access types to address static objects in embedded real-time applications. Certain Ada compilers support only limited types of parameters when pragma INTERFACE is used (e.g., to assembly language, C, or some other language selected for efficiency or interface requirements), and access types are usually one of those supported when other static types, such as record structures, are not.

b) If memory considerations are a concern, are short-circuits used over IF statements?

c) Has overhead execution time been minimized for procedure calls? A potential run-time efficiency advantage of blocks is memory savings. If a declare statement is used in the block, the resources for data which are local to the block will only be allocated when the block is entered. Additionally, some compilers may deallocate these memory resources after the block is exited.

d) Has the use of Exit's been avoided? Exit's should not be used because they can greatly increase run time if the code at the exit address has to be paged into working space.

e) Is the pragma INLINE used where call overhead is of paramount concern?

f) Has suppression of exception checks been minimized during operation?

g) Is pragma SHARED used only when forced by run time system deficiencies?

h) Have hardware and implementation dependencies been encapsulated in a package or packages?

i) Have the objectives been clearly indicated if machine or solution efficiency is the reason for hardware or implementation dependent code?

j) Have interrupt receiving tasks been isolated into implementation dependent packages?

k) Has the DELAY statement been used with caution when timing accuracy is essential?

B1.3 Software Assessment Criteria

Software criteria focuses on providing guidance in developing reusable software components.

B1.3.1 Identifiers

Description

The naming and declarations of constants, variables, subprograms, etc., are inspected in this section. Appropriate use of constants is also checked.

Items

a) Are application-independent but meaningful (application specific) names used for the reusable part and its identifiers? (Mandatory)

b) Have abbreviations in identifier or unit names been avoided? (The only exception is that an abbreviated format for a fully qualified name can be established via the RENAMES clause.)

c) Have the predefined numeric types in package STANDARD been avoided?

d) Are literal expressions represented in a radix appropriate to the reusable part? (Decimal and octal numbers should be grouped by three's counting from each side of the radix point. Hexadecimal numbers should be grouped by four's counting from each side of the radix point.)

e) Are pragmas and attributes added by the implementor avoided?

f) Are the package SYSTEM constants only used when generalizing other machine dependent constructs? (Mandatory, since the values in this package are implementation-provided, unexpected effects can result from their use.)

B1.3.2 Local Programming Practices

Description

The source text is inspected for compliance with programming practices requirements and known specific problems in program logic.

Items

a) Are default values provided for any parameters not included in the original part?

b) Is the size of local variables dependent on the actual parameter size where possible?

c) Has reusable code been structured so that the compiler can exploit dead code removal where it is supported by the implementation?

d) Is dependence on the order in which Ada constructs are evaluated avoided?

   - If a subprogram changes the value of an actual parameter, are there no instances where a variable and a call to the subprogram with the variable as the actual parameter are evaluated as part of the same expression?

e) Are specifications of packages and their bodies separated? (Mandatory.)

f) Are relational tests done with "<=" and ">=" instead of "<",">", "=", and "/="? (Mandatory.)

g) Is "=" never used on real operands as a condition to exit a loop?

h) Is the depth of nested expressions and control structures restricted to 5? (This number can be modified for each program. It is suggested that the maximum number of nesting levels be kept between three and five.)

i) Are values of type attributes used in comparisons and checking for small values?

j) Are exception handlers provided for CONSTRAINT_ERROR and NUMERIC_ERROR? (Recommended. Either of these exceptions may be raised and exception handlers should be prepared to handle either.)

k) Are implementation-defined exceptions avoided?

l) Is the use of representation clauses isolated? (The Ada LRM does not require that these clauses be supported for all types. Therefore, isolating representation clauses will minimize the impact of any changes necessitated by a port. The two exceptions to this guideline are for task storage size and access collection size, where portability may be enhanced through their use.)

m) Are default parameters provided for special usage subprograms and entries?

n) Are default parameters placed at the end of the formal parameter list?

o) Are long fully qualified names renamed to reduce complexity? (Desired, but not mandatory. An abbreviated format for a fully qualified name can be established via the RENAMES clause. This capability is useful when a very long fully qualified name would otherwise occur many times in a localized section of code.)

p) Are declarations, renamed for visibility purposes, used instead of using the USE clause?

q) Are 'RANGE, 'FIRST, 'LAST, 'LENGTH used instead of constants or variables where appropriate to access array elements? (Desired, but not mandatory.)

r) If 'RANGE, 'FIRST, 'LAST, 'LENGTH is used, is there testing for null arrays?

s) Are range declarations used instead of INTEGER types where range values are known?

t) Are indices of arrays declared as a named discrete type?

u) Are named types used instead of anonymous types? (Although Ada allows anonymous types, they have limited usefulness and complicate program modification. For example, a variable of anonymous type can never be used as an actual parameter because it is not possible to define a formal parameter of the same type. Even though this may not be a limitation initially, it precludes a modification in which a piece of code is changed to a subprogram. Also, two variables declared using the same anonymous type declaration are actually of different types.)

v) Is data initialization located in the declarative part of the subprogram unit?

w) Declare parameters in a consistent order. A suggested order is:

- All IN parameters without default values are declared before any IN OUT parameter.
- All IN OUT parameters are declared before any OUT parameters.
- All parameters with default values are declared last.
- The order of parameters within these groups is derived from the needs of the application.

x) Are data parameters placed before control parameters (options)?

B1.3.3 Tasking Semantics

Description

This section checks for the correct and best use of task objects.

Items

a) When declared in the same declarative list, is the order in which task objects are activated not dependent upon?

b) Is a representation clause used to identify the expected stack requirements for each task? This is an acceptable exception to the guideline, "Avoid the use of representation clauses."

c) Are timing requirements dependent on the pattern of execution task?

d) Do the tasking semantics of the program avoid dependence on a specific value of a delay?

e) Are busy waiting loops never used instead of a delay?

f) Is polling limited to those cases where it is absolutely necessary?

g) Is only the pragma PRIORITY used to distinguish general levels of importance?

h) Has dependence upon the order in which guard conditions are evaluated or on the algorithm for choosing among several open select alternatives been avoided?

i) Has sharing of variables between tasks been avoided?

j) Do tasks communicate only through the rendezvous mechanism?

k) Have shared variables as a task synchronization device been avoided?

l) Is the pragma SHARED used only when forced by run time system deficiencies?

m) Is the ABORT statement avoided?

n) Are interrupts passed to the main tasks via a normal entry?

o) Are task specifications and package specifications provided as separate entities?

B1.3.4 Application of Generics

Description

This section checks for the correct and best use of generics.

Items

a) Are iterators provided for traversing complex data structures within reusable parts?

b) Are families of generic or other parts with similar specifications created?

c) Is named association used when instantiating generics with many formal parameters?

d) Is RENAME used instead of the USE clause to facilitate modification?

e) In generic units with a lot of subprogram parameters, are default parameters of null subprograms provided?

B1.3.5 Global Programming Practices

Description

The source text is inspected for compliance with programming practice requirements involving a group or package of reusable parts.

Items

a) Are symbolic constants and constant expressions used to allow multiple dependencies to be linked to a single or small number of symbols?

b) Are exceptions propagated out of reusable parts?

c) Have dependencies on the exact locations, at which predefined exceptions are raised, been avoided?

d) Has the use of non-standard character sets been avoided?

e) Is the main program procedure free of parameters?

f) Are hardware and implementation dependencies encapsulated in a package or packages?

g) Have interfaces with other languages been isolated?

h) Are all subprograms employing the pragma INTERFACE to an implementation dependent (interface) package isolated?

APPENDIX C
REUSE LIBRARY CRITERIA

C1.1 What is the approach to component classification?

(At present, no single classification scheme has been determined to be the definitive method. Each organization setting up a reuse library will have to do an analysis to determine which scheme is appropriate for their needs.)

C1.2 Is the library information model well-defined and adaptable?

The library information model should fully describe the objects, attributes, and relationships of the reuse library system.

C1.3 Does the library tool set provide needed capability?

To provide a library that is easy to use and efficient for the developer, the user interface must be tailorable to the selected software domain. This involves the separating of functions of the library from the functions surrounding the use of the library and presenting the user with a view of the library in domain specific terms. The vocabulary created in the domain analysis task is used in customizing the interface.

Each organization should determine if the library has the following set of associated tools or capability: search, file browser, extraction, supply, catalog, problem reporting, and configuration management system.

C1.4 Does the library support multiple domains?

C1.5 Is there a library search tool?

A library search tool provides for searching of the library based on the classification scheme, keywords and attributes.

C1.6 Is there a file browser?

A file browser allows the user to look at the component that has been retrieved.

C1.7 Is there an extraction tool?

An extraction tool allows the user to copy an asset from the reuse library to be reused.

C1.8 Is there a supply?

A supply is an interactive process that allows users to submit software for inclusion in the library.

C1.9 Is there a catalog function?

This function provides the capability to create an updated catalog of abstracts for each component in the library.

C1.10 Is there a problem reporting system?

A problem reporting system allows the users of the library to report problems with the library system or with the components retrieved.

C1.11 Is there a configuration management system?

To be able to manage a large number of files with multiple versions, the library should provide a configuration management capability or provide an interface to a configuration management system.

C1.12 Does the library accommodate a continually expanding collection of components?

There should be no limitation to the number of components submitted to the library other than the physical limitations of storage space of the library hardware.

C1.13 Does the library support finding similar components, not just exact matches?

In addition, the library should have the means whereby the user can broaden or narrow the search automatically.

C1.14 Does the library support a domain vocabulary and class structure which can be very precise and have a high descriptive power?

Both are necessary conditions for classifying and cataloging software.

C1.15 Does the library support a thesaurus capability?

This is highly desirable for libraries which have a large number of infrequent users.

C1.16 Is it easy to maintain (i.e., add, delete, and update) the class structure and vocabulary without need for reclassification of existing components?

This is highly desirable for libraries with a large number of components.

C1.17 Is the library easily usable by both the librarian and the end user?

Ease of use for both is highly desirable, however, if the ease of use favors one over the other, it should be toward the end user. The end user may be a one-time or infrequent user, where ease of user is critical. Since the librarian will be using the library daily, ease of use is not as critical.

C1.18 Does the library support the capability to specify the severity of the testing to which the material has been subjected in its current form?

C1.19 Does the library provide the ability to export its classification scheme and catalog into a documented external representation? And, conversely, does the library provide the ability to import information from this external representation into its classification scheme and catalog?

A library that can import and export classification and catalog information into a documented external representation can exchange that information with other libraries.

C1.20 Does the library support tool integration?

The tools in a project's development environment can more effectively support the reuse process if they have access to the information maintained by the reuse library.

American Institute of Aeronautics and Astronautics

370 L'Enfant Promenade, SW
Washington, DC 20024-2518

ISBN 1-56347-049-7